Noise-contrastive estimation of normalising constants and GANs

Contents

1 Fonctions génériques

1.1 Algorithme d'Hasting

Utilité : simuler selon $p_m(.,\psi)$ pour un paramètre ψ choisi.

Argument	**Type**	**Exemple**	**Indication**
x	vecteur	rcauchy(100, 0, 1)	notre échantillon de densité inconnue p_m
n	entier	100	taille de la simulation
psi	vecteur	c(0,1)	paramètres choisis de la fonction p_m
h	fonction		fonction qui retourne $\overline{p_m}(.,\psi)$

```
hasting = function(x, n, psi, h){
  y = c()
  y = append(y, sample(x, 1))
  for (i in 2:n){
    y_ = rnorm(1, y[i-1], 1)
    u = runif(1)
    if ( u <=
         (h(y_,psi) * dnorm(y_, y[i-1], 1))
         /(h(y[i-1],psi) * dnorm(y[i-1], y_, 1))
    ){
      y = append(y, y_)}
    else {
      y = append(y, y[i-1])
    }
  }
  return (y)
}
```

Ci-dessous une autre version qui génère un échantillon iid.

Argument	**Type**	**Exemple**	**Indication**
x	vecteur	rcauchy(100, 0, 1)	notre échantillon de densité inconnue p_m
n	entier	100	taille de la simulation
psi	vecteur	c(0,1)	paramètres choisis de la fonction p_m
h	fonction		fonction qui retourne $\overline{p_m}(.,\psi)$
ϵ	Entier	2	pas de décorrélation

```
hasting_iid = function(x, n, psi, h, eps){
  y = c()
  y = append(y, sample(x, 1))
  for (i in 2:(n*eps)){
    y_ = rnorm(1, y[i-1], 1)
    u = runif(1)
    if ( u <=
         (h(y_,psi) * dnorm(y_, y[i-1], 1))
         /(h(y[i-1],psi) * dnorm(y[i-1], y_, 1))
    ){
      y = append(y, y_)}
    else {
      y = append(y, y[i-1])
    }
  }
  filter = y * rep(c(1,rep(0, eps-1)), n)
```

```
  return (filter[filter != 0])
}
```

1.2 MC MLE (Geyer)

Utilité : retourne une estimation des paramètres selon la méthode décrite dans le papier de Geyer.

```
mc_mle = function(x, n, psi, h){

  m = length(x)

  y = hasting(x, n, psi, h)

  L = function(theta){
    return(sum(log(h(x,theta)/h(x,psi))) - m*log(mean(h(y,theta)/h(y,psi))))
  }

  theta = optim(
    par = rep(1,length(psi)),
    gr = "CG",
    control = list(fnscale=-1),
    fn = L
  )$par

  return(theta)
}
```

1.3 Reverse logistic regression

La maximisation de la fonction objectif

$$l_n(\eta) = \sum_{j=1}^{m}\sum_{i=1}^{n_j} log(p_j(X_{i,j}, \eta))$$

permet d'estimer les η qui sont fonction des constantes de normalisation des h_j. On utilise les notations suivantes :

$$\eta_j = -log(Z_j) + log(\frac{n_j}{n}) \text{ avec } Z_j \text{ la constante de normalisation de } h_j$$

$$p_j(x) = \frac{h_j(x)e^{\eta_j}}{\sum_{k=1}^{m} h_k(x)e^{\eta_k}}$$

Première approche avec $m = 2$, $n = n_1 + n_2$. On note $x = (X_{1,1}, ..., X_{n_1,1})$ l'échantillon de données de loi $\frac{h(,\alpha^*)}{Z_1(\alpha^*)}$, avec Z_1 que l'on cherche à estimer et α^* estimé par MCMLE. On note $y = (X_{1,2}, ..., X_{n_2,2})$ l'échantillon de données $\frac{h(,\psi)}{Z_2(\psi)}$ avec ψ que l'ont choisit. x est donné, on simule y avec une version *iid* de l'algorithme d'Hasting.

```
rev_log_reg = function(x, alpha, psi, n, h, eps){

  m = length(x)
  y = hasting_iid(x, n, psi, h, eps)

  # calcul des probabilités p_j
  denom = function(sample, eta) {
```

Argument	Type	Exemple	Indication
x	vecteur	rcauchy(100, 0, 1)	notre échantillon de densité $p_m(.,\alpha^*)$
alpha	vecteur	c(0,1)	paramètres de la fonction p_m, estimés par MC MLE
psi	vecteur	c(0,1)	paramètres choisis de la fonction p_m
n	entier	100	taille de l'échantillon de densité $p_m(.,\psi)$
h	fonction		fonction qui retourne $\overline{p_m}(.,.)$
ϵ	Entier	2	pas de décorrélation pour *hasting iid*

```
    return(pm_barre(sample, alpha)*exp(eta[1]) + pm_barre(sample,psi)*exp(eta[2]))}
  p_1 = function(sample, eta){
    return (pm_barre(sample, alpha)*exp(eta[1]) / denom(sample, eta))}
  p_2 = function(sample, eta){
    return (pm_barre(sample, psi)*exp(eta[2]) / denom(sample, eta))}

  # fonction objectif
  L = function(eta) {
    return(sum(log(p_1(x, eta))) + sum(log(p_2(y, eta))))}

  # initialisation descente de gradient
  eta1 = -log(sd(x)*sqrt(2*pi)) + log(m/(m+n))
  eta2 = -log(sd(y)*sqrt(2*pi)) + log(n/(m+n))

  # optimisation
  const = optim(
    par = c(eta1,eta2),
    gr = "CG",
    control = list(fnscale=-1),
    fn = L
  )$par

  a = exp(-const[1] + log(m/(m+n)))
  return(a)
}
```

Seconde approche avec $m = 2$, $n = n_1 + n_2$. On note $x = (X_{1,1}, ..., X_{n_1,1})$ l'échantillon de données de loi $\frac{h_1(,\alpha^*)}{Z_1(\alpha^*)}$, avec Z_1 que l'on cherche à estimer et α^* estimé par MCMLE. On note $y = (X_{1,2}, ..., X_{n_2,2})$ l'échantillon de données $h_2(,\psi)$ avec ψ que l'ont choisit et h_2 une loi usuelle dont on connait la constante de normalisation. x est donné, on simule y avec les méthodes usuelles.

Argument	Type	Exemple	Indication
x	vecteur	rcauchy(100, 0, 1)	notre échantillon de densité $p_m(.,\alpha^*)$
alpha	vecteur	c(0,1)	paramètres de la fonction p_m, estimés par MC MLE
psi	vecteur	c(0,1)	paramètres choisis d'une loi usuelle p_n
n	entier	100	taille de l'échantillon de densité p_n
h1	fonction		fonction qui retourne $\overline{p_m}(.,\alpha^*)$
h2	fonction		fonction qui retourne $p_n(.,\psi)$
law_y	fonction	rnorm	fonction qui retourne un échantillon suivant la loi p_n

```
rev_log_reg_with_noise = function(x, alpha, psi, n, h1, h2, law_y){

  y = do.call(law_y, c(list(n),psi))
  m = length(x)
```

```r
  # calcul des probabilités p_j
  denom = function(sample, eta) {
    return(h1(sample, alpha)*exp(eta[1]) + h2(sample,psi[1],psi[2])*exp(eta[2]))}
  p_1 = function(sample, eta){
    return (h1(sample, alpha)*exp(eta[1]) / denom(sample, eta))}
  p_2 = function(sample, eta){
    return (h2(sample, psi[1],psi[2])*exp(eta[2]) / denom(sample, eta))}

  # fonction objectif
  L = function(eta) {
    return(sum(log(p_1(x, eta))) + sum(log(p_2(y, eta))))}

  # initialisation descente de gradient
  eta1 = -log(sd(x)*sqrt(2*pi)) + log(m/(m+n))
  eta2 = -log(sd(y)*sqrt(2*pi)) + log(n/(m+n))

  # optimisation
  const = optim(
    par = c(eta1,eta2),
    gr = "CG",
    control = list(fnscale=-1),
    fn = L
  )$par

  b = exp(-const[2] + log(m/(m+n)))
  a = exp(-const[1] + log(m/(m+n)))

  # la constante de h_2 vaut normalement 1 si h_2 est une loi de densité, donc b est exactement le coef
  return(a/b*m/n)
}
```

1.4 NCE (Gutmann)

Utilité : Retourne l'estimation de la constante et des paramètres.

Argument	Type	Exemple	Indication
x	vecteur	rcauchy(100, 0, 1)	notre échantillon de densité inconnue
law_y	fonction	rnorm	fonction qui retourne un échantillon suivant la loi p_n
n	entier	100	taille de l'échantillon de bruit suivant la loi p_n
params_y	vecteur	c(0,1)	arguments de la fonction law_y
log_pm	fonction		fonction qui retourne le logarithme de la densité p_m
log_pn	fonction		fonction qui retourne le logarithme de la densité p_n
size_theta	entier	3	taille de θ, vaut habituellement 2 ou 3

```r
nce = function(x, law_y, params_y, log_pm, log_pn, size_theta, n){

  y = do.call(law_y, c(list(n),params_y))

  m = length(x)

  h = function(u, theta){
    return( 1 / (1 + n/m * exp(log_pn(u) - log_pm(u, theta))))
  }
```

```r
  J = function(theta){
    return( sum(log(h(x, theta))) + sum(log(1 - h(y, theta))) )
  }

  theta = optim(
    par = rep(1, size_theta),
    gr = "CG",
    control = list(fnscale=-1),
    fn = J
  )$par

  return(c(theta[-size_theta], exp(-theta[size_theta])))
}
```

1.5 Graphiques

Utilité : afficher l'histogramme pour un échantillon de données x.

```r
print_hist = function(x) {
  df = data.frame(x = x)
  hist_x = ggplot(df, aes(x=x)) +
           geom_histogram(aes(y = stat(count)/sum(count)), bins = 20, color="white") +
           theme(aspect.ratio = 1) +
           labs(y = "Fréquence") +
           ggtitle("Distribution de l'échantillon x")
  print(hist_x)
}
```

Utilité : pour NCE, afficher l'évolution des paramètres au fur et à mesure de l'augmentation de n (la dimension de l'échantillon de bruit)

```r
NCE_evol_params = function(x, law_y, params_y, log_pm, log_pn, size_theta, ratio, steps, labels) {

  # Creation de l'abscisse
  m = length(x)
  N = seq(0, m*ratio, length.out = steps + 1)

  # Creation de l'ordonnée
  theta = c()
  for (n in N) {
    theta = append(theta, nce(x, law_y, params_y, log_pm, log_pn, size_theta, n))
  }

  # Formatage des données
  theta = t(rbind(matrix(theta, nrow = size_theta),N))
  df = as.data.frame(theta)
  df_melted = melt(df, id.vars = "N")

  # Plot
  plot_df = ggplot(df_melted, aes(x = N, y = value)) +
  geom_line(aes(color = variable, group = variable)) +
  geom_point(aes(color = variable, group = variable)) +
  labs(title = "Evolution des paramètres par rapport au bruit",
       x = "n (taille du bruit)",
```

```
      y = "Paramètres",
      color = "Légende") +
  scale_color_manual(labels = labels, values = c("blue", "red", "orange"))

  print(plot_df)

  #return(theta)
}
```

Note : il faudrait optimiser le temps de calcul de ces fonctions, peut-être en matriciel au lieu des boucles ou bien avec du calcul en parralèle sur CPU/GPU

2 Applications

2.1 Exemple basique : la loi normale

Soit x l'échantillon de taille m obtenu selon la loi de densité inconnue p_d.

On considère ici que p_d appartient à la famille de fonctions paramétrées par $\theta = (c, \mu, \sigma)$ suivante :

$$p_m(u;\theta) = \frac{1}{Z(\mu,\sigma)} \times exp\Big[-\frac{1}{2}\big(\frac{u-\mu}{\sigma}\big)^2\Big] \quad \text{d'où} \quad ln(p_m(u;\theta)) = c - \frac{1}{2}\big(\frac{u}{\sigma} - \frac{\mu}{\sigma}\big)^2$$

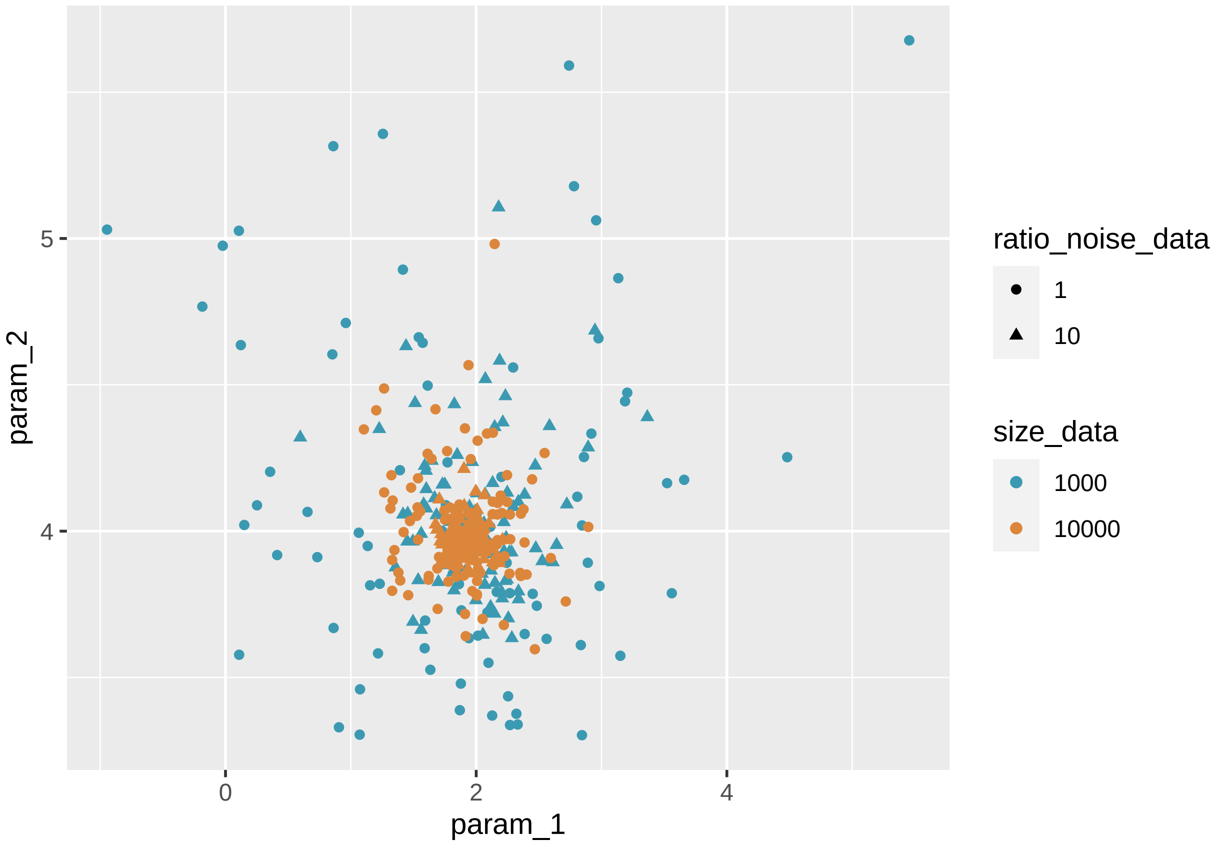
```
pm_barre = function(u, theta){
  return(exp(-0.5 * ((u - theta[1]) / theta[2]) ** 2))
}

log_pm = function(u,theta){
  return(theta[3] - 1/2 * (u/theta[2] - theta[1]/theta[2]) ** 2)
  # theta[1] = mu / theta[2] = sigma / theta[3] = c
}

log_pn_cauchy = function(u){
  return(log(dcauchy(u, mean(x), sd(x))))
}

m = 10000
n = 10000
x = rnorm(m, 2, 4)
size_theta = 3
```

```
# METHODE MC MLE
mc_mle(x, n, c(mean(x),sd(x)), pm_barre)
```

```
## [1] 2.454480 3.633104
```

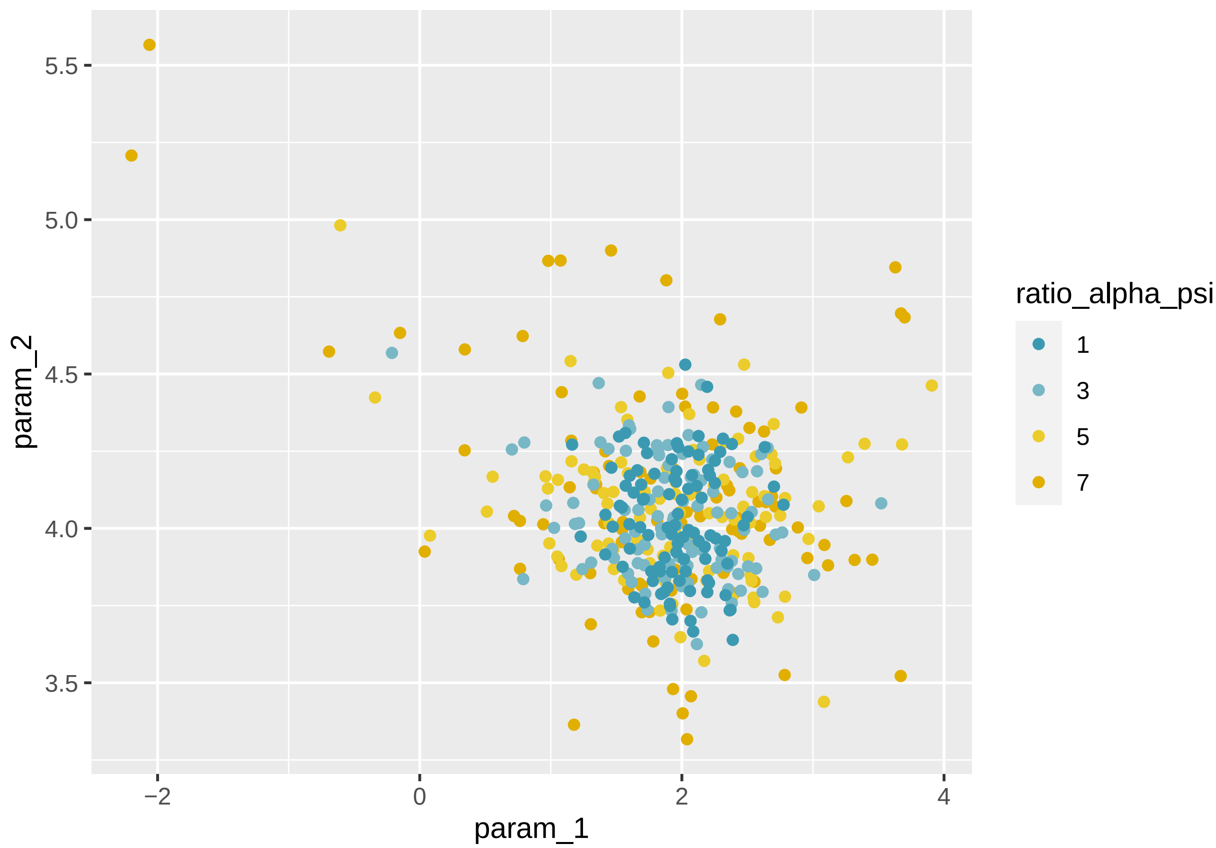
Etudions l'impact de la dimension des échantillons sur la convergence des estimateurs.

```
ggplot(df_mcmle_filt, aes(x = param_1, y = param_2, color = size_data, shape = ratio_noise_data)) + geom
```



Etudions l'impact du choix de ψ sur la convergence des estimateurs.

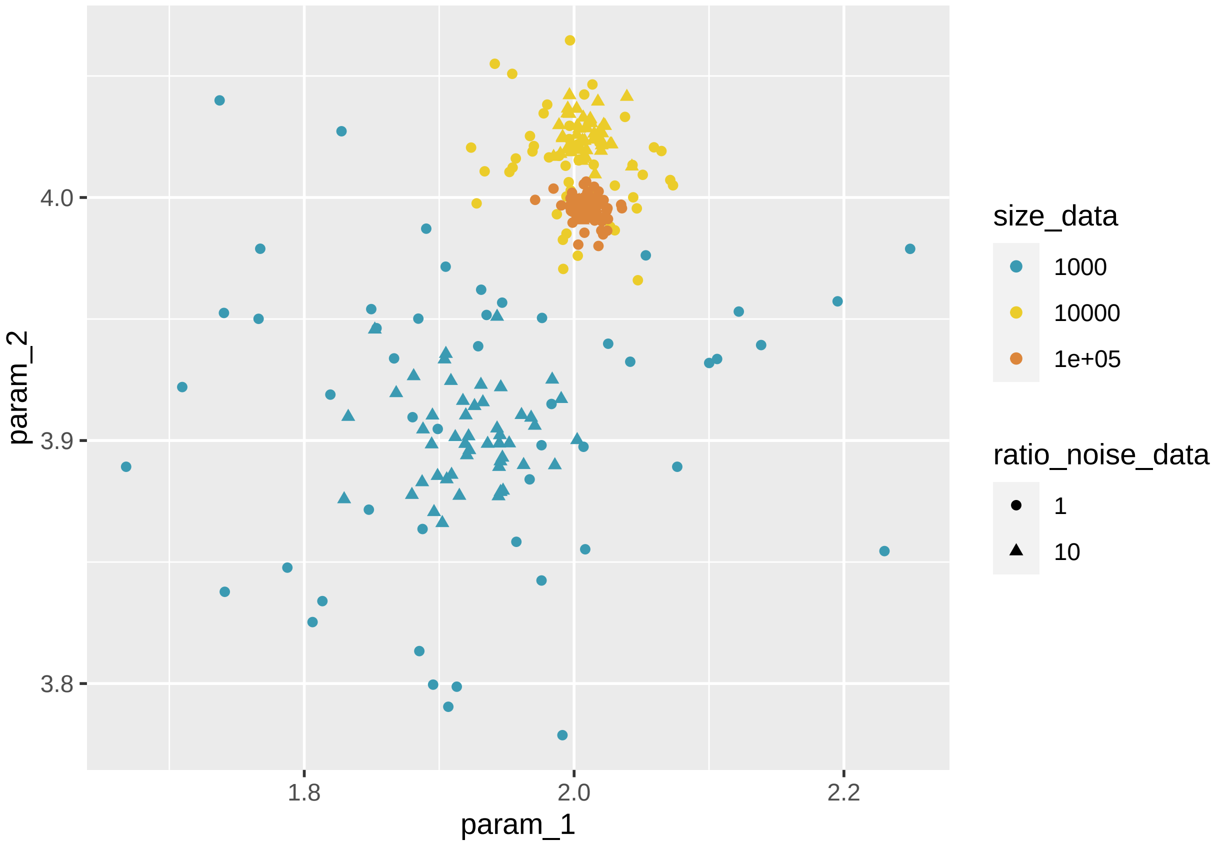
```
df_mcmle_psi <- read_csv("df_mcmle_psi.csv")[,-1]

df_mcmle_psi = df_mcmle_psi[order(-df_mcmle_psi$ratio_alpha_psi),]

df_mcmle_psi$ratio_alpha_psi = as.factor(df_mcmle_psi$ratio_alpha_psi)
ggplot(df_mcmle_psi, aes(x = param_1, y = param_2, color = ratio_alpha_psi)) + geom_point() + scale_colo
```



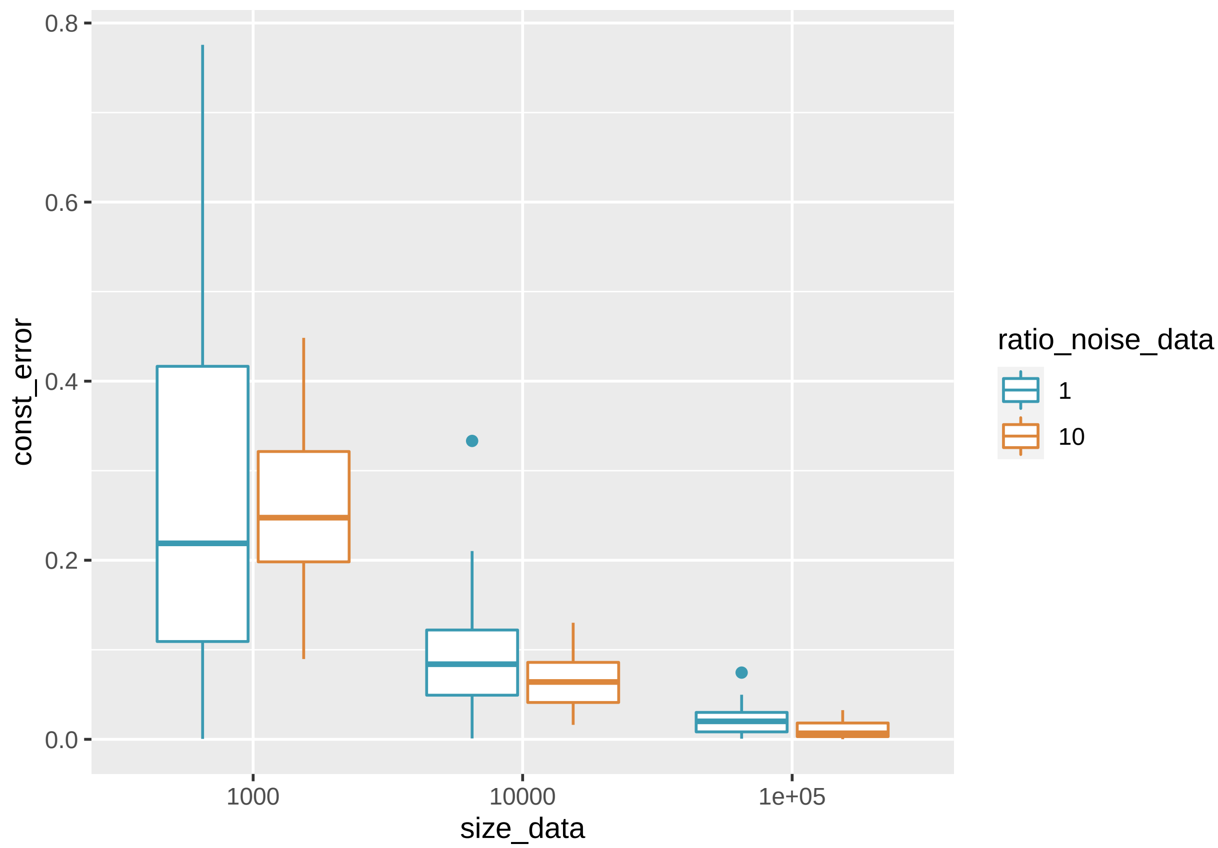

```
# METHODE NCE
nce(x, rcauchy, c(mean(x),sd(x)), log_pm, log_pn_cauchy, size_theta, n)
```

```
## [1] 2.014970 3.964599 9.962068
```

```
ggplot(df_nce, aes(x = param_1, y = param_2, color = size_data, shape = ratio_noise_data)) + geom_point
```

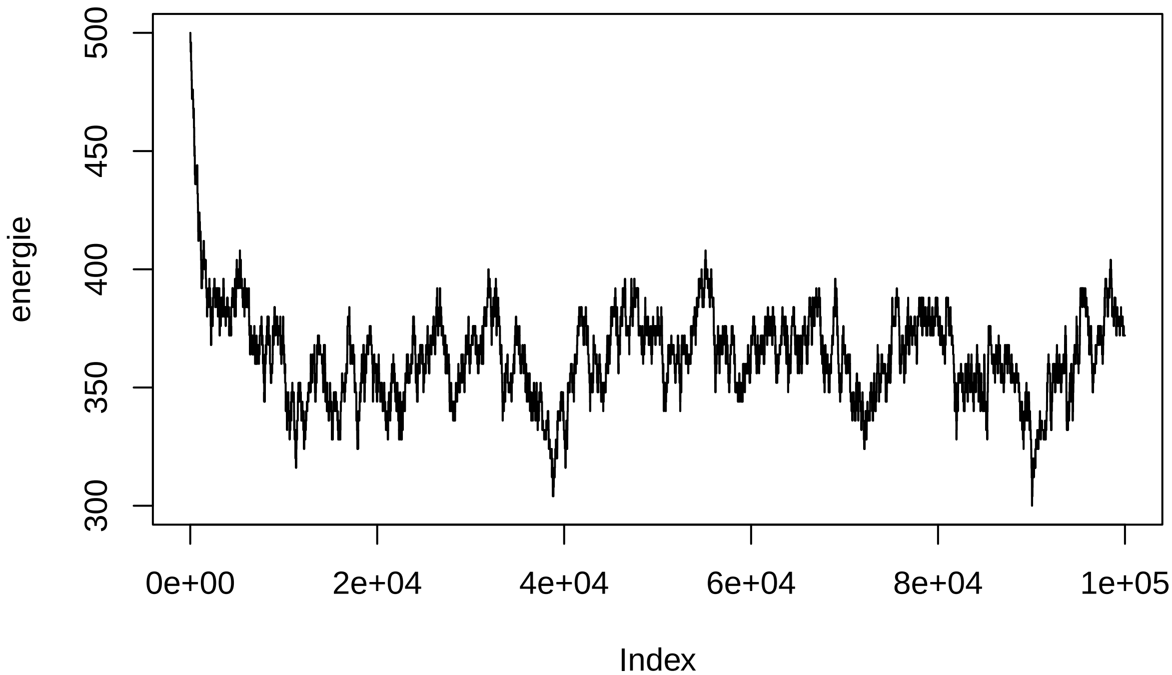



```
ggplot(df_nce, aes(x = size_data, y = const_error, color = ratio_noise_data)) + geom_boxplot() + scale_c
```



2.2 Modèle d'Ising

```
# temps estimé 1min n=500, iter=100k
library('isingLenzMC')

beta    = 0.9              # paramètre de temperature
n       = 500              # nombre de sites
config  = rep(1,n)         # generer une configuration à n sites
config1 = genConfig1D(n)   # generer une configuration à n sites aléatoirement

energie = c()
iter = 100000
data = matrix(nrow= iter, ncol = n)

for (k in 1:iter){
  # on tire successivement des configurations jusqu'à obtenir convergence du niveau d'énergie
  #ce que l'on observera graphiquement
  config = isStep1D(beta, config, 1.0, 0.0, 1)$vec # tirage avec Metropolis
  data[k,] = config
  energie   = c(energie,totalEnergy1D(config,1.0,0.0))
}
plot(energie, type ='l')
```



On définit la fonction *coeur* qui est à une constante de normalisation près la mesure de Gibbs associée à au modèle d'Ising de paramètre β. On définit la fonction $ising1D(n)$ qui est rend toutes les configurations de spins possibles d'un modèle d'Ising unidimensionnel à n sites.

```
coeur = function(constante_normalisation,beta=0.9,config){

  # constante_normalisation :  fonction de partition inconnue
  # beta                    :  paramètre de température
  # config                  :  une configuration de spins

  # Lorsque 'constante_normalisation' == vraie valeur de la fonction de partition pour le paramètre bet
  #        alors return la probabilité de la configuration de spins 'config'

  return ( exp(-beta*totalEnergy1D(config,1.0,0.0))/constante_normalisation )
}

ising1D = function(n){

  # n : nombre de sites pour une configuration

  # return l'ensemble des configuration 1D possibles dans une matrice
  # chaque ligne correspond à une configuration possible

  if (n==1)
    {
    return(matrix(c(1,-1),nrow=2))
    }
```

```r
  else
    {
    return(cbind( rbind(ising1D(n-1),ising1D(n-1)), c(rep(1,2**(n-1)),rep(-1,2**(n-1)))  ))
    }
}
```

```r
# un petit test de ce qu'on fait

n       = 15   # nombre de sites
config  = ising1D(n) # ensemble des configurations 1D à n sites
beta    = 0.9 # paramètre de température
energie = c()

for (k in 1:2**n){ energie = c(energie,totalEnergy1D(c(config[k,]), 1, 0) ) } # totalEnergy1D(config[k,

cst_normalisation  = sum(exp(-beta*energie))

proba  = 0
for (k in 1:2**n){ proba = proba + coeur(cst_normalisation,beta=0.9,config[k,]) }
(proba)
print("if proba = 1, then it is good")
```

Il se peut qu'il y ait conflit de notations pour cette sous section d'Ising avec les notations du précédent exemple. De plus, il y'a une erreur en sortie que je n'arrive pas résoudre pour l'instant. Mais je ne pense pas qu'elle soit très compliquée...

```r
log_pn = function(u,param_pn){
  # pn densite iid N(param_pn[1],param_pn[2]) de dim length(u)
  # Verifier que c'est bien ce qui est codé
  n     = length(u)
  log_d = 0
  for (k in 1:n){ log_d = log_d - 0.5 * ((u[k] - param_pn[1]) / param_pn[2]) ** 2 }
  return(-0.5*n*log(2*pi*(param_pn[2]**2)) + log_d)
}

log_pm = function(configuration,beta,theta){
  # theta = -log(Z)
  # interraction entre sites = 1 et avec le champs extérieure = 0
  return ( -beta*totalEnergy1D(configuration,1,0) + theta )
}


nce_ising = function(matrix_ising,beta,log_pm,theta,law_pn,param_pn,log_pn,b_size){
  # matrix_ising = matrice ou en ligne sont rangées les observations d'Ising en 1D
  # beta         = paramètre de température de modèle d'Ising \in ]0,1[
  # log_pm       = logarithme de la densité d'Ising
  # theta        = paramètre à optimiser , en lien avec la constante de normalisation

  # law_pn   = loi du bruit qui doit contenir dans son support {0,1}
  # param_pn = paramètres de la loi law_pn
  # log_pn   = log de la densité de pn

  # b_size  = nombre d'observation du bruit, à ne pas confondre avec la taille d'une observation du bru

  n     = dim(matrix_ising)[2] # n = nombre de sites pour les observations d'Ising en 1D
```

```r
  m     = dim(matrix_ising)[1] # m = nombre d'observations d'Ising en 1D

  bruit = matrix(do.call(law_pn,c(list(b_size*n),param_pn)),ncol=n) # échantillon de vecteurs de bruit

  h = function(configuration,theta){return( 1 / (1 + b_size/m * exp(log_pn(configuration,param_pn) - log

  J = function(theta){
    # x vecteur de densite inconnue
    # y vecyeur de bruit
    objectif = 0
    for (k in 1:m){objectif = objectif + log(h(matrix_ising[k,],theta)) }   # composante de l'objectif
    for (k in 1:b_size){objectif = objectif + log(1 - h(bruit[k,],theta)) } # composante de l'objectif
    return( objectif )
    }

  solution = optimize(f = J,
                      interval = c(-1e5,1e5),
                      maximum  = FALSE)

  return(exp(-solution))
}


n            = 10
matrix_ising = ising1D(n)[100:900,]
beta         = 0.9
theta        = 1
law_pn       = rnorm
param_pn     = c(0,1)
b_size       = 2**n

nce_ising(matrix_ising,beta,log_pm,theta,law_pn,param_pn,log_pn,b_size)
```

3 Nouvelles approches

3.1 Bootstrap

```
# Calcul de {size_boot} estimateurs par bootstrap
NCE_bootstrap = function(x, law_y, params_y, log_pm, log_pn, size_theta, n, size_boot, labels) {
  m = length(x)
  theta_bootstrap = c()
  x_bootstrap = x
  for (i in 1:size_boot) {
    theta_bootstrap = append(theta_bootstrap, nce(x_bootstrap,
                                              law_y,
                                              params_y,
                                              log_pm,
                                              log_pn,
                                              size_theta,
                                              n))
    x_bootstrap = sample(x, size = m, replace=TRUE)
  }
  return(matrix(theta_bootstrap, nrow = size_theta))
}

# Plot la moyenne empirique des estimateurs bootstrap en fonction du nombre d'estimateurs
NCE_bootstrap_plot = function(matrix_theta_bootstrap) {

  # Formatage des données pour plot
  array_boot = 1:length(matrix_theta_bootstrap[1,])
  df = as.data.frame(cbind(t(rowCumsums(matrix_theta_bootstrap))/array_boot,array_boot))
  df_melted = melt(df, id.vars = "array_boot")

  # Plot
  plot_df = ggplot(df_melted, aes(x = array_boot, y = value)) +
  geom_line(aes(color = variable, group = variable)) +
  labs(title = "Evolution des paramètres par bootstrap",
       x = "Taille du bootstrap",
       y = "Paramètres",
       color = "Légende") +
  scale_color_manual(labels = labels, values = c("blue", "red", "orange"))
  print(plot_df)
}


# etude bootstrap de l'estimateur
bootstrap = function(matrix, alpha){
  return(data.frame(
    theta = matrix_theta_bootstrap[,1],
    biais = rowMeans(matrix_theta_bootstrap) - matrix_theta_bootstrap[,1],
    IC = rowQuantiles(matrix_theta_bootstrap, probs = c(alpha/2, 1-alpha/2))
  ))
}
```

```
x_test = rnorm(1000,2,4)

matrix_theta_bootstrap = NCE_bootstrap(x_test, rcauchy, c(mean(x_test),sd(x_test)), log_pm, log_pn_cauc
```

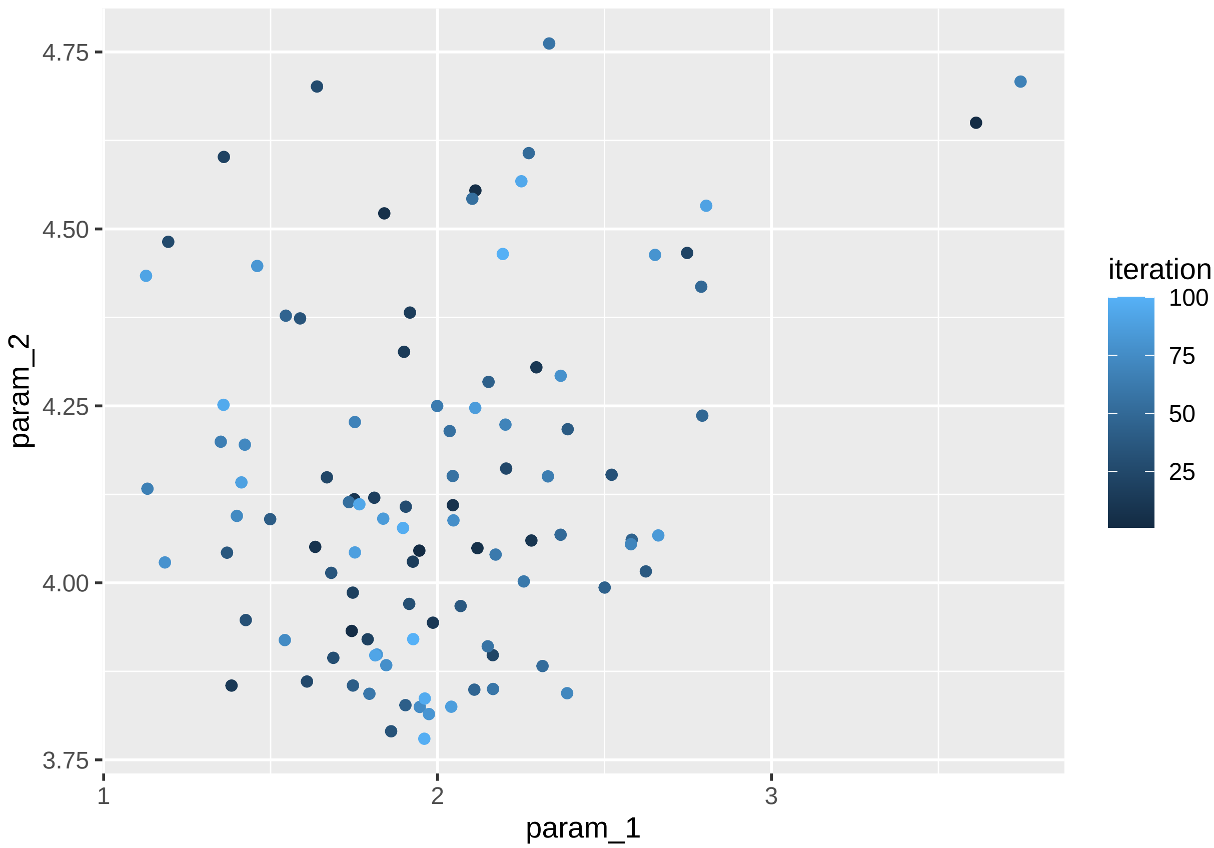
```
kable(bootstrap(matrix_theta_bootstrap, 0.05))
```

theta	biais	IC.2.5.	IC.97.5.
2.036242	0.0617818	1.873479	2.408742
3.978383	-0.0348073	3.820189	4.014394
10.070408	-0.2162332	9.303478	10.187756

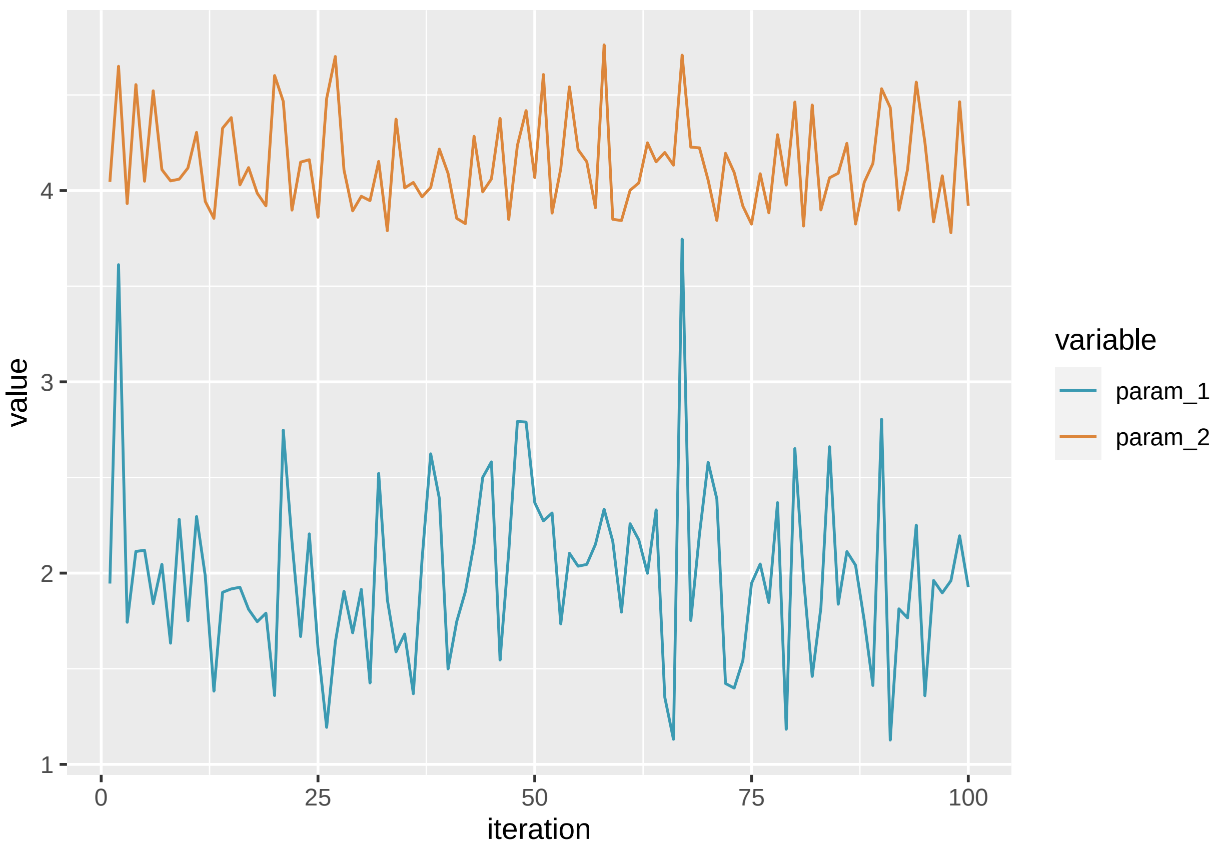
3.2 Récursivité

Utilité : améliorer récursivement la précision de l'estimation via les estimations précédentes

```
df_recurs_naif <- read_csv("df_recurs_naif.csv")[,-1]
ggplot(df_recurs_naif, aes(x = param_1, y = param_2, color = iteration)) + geom_point()
```




```
df_recurs_naif <- read_csv("df_recurs_naif.csv")[,-1]
colnames(df_recurs_naif) = c("iteration", "param_1", "param_2")

df_recurs_naif_melted = melt(df_recurs_naif, id.vars = "iteration")
ggplot(df_recurs_naif_melted, aes(x = iteration, y = value)) + geom_line(aes(color = variable, group = v
```



Cette approche naïve n'améliore pas la précision de notre estimation.

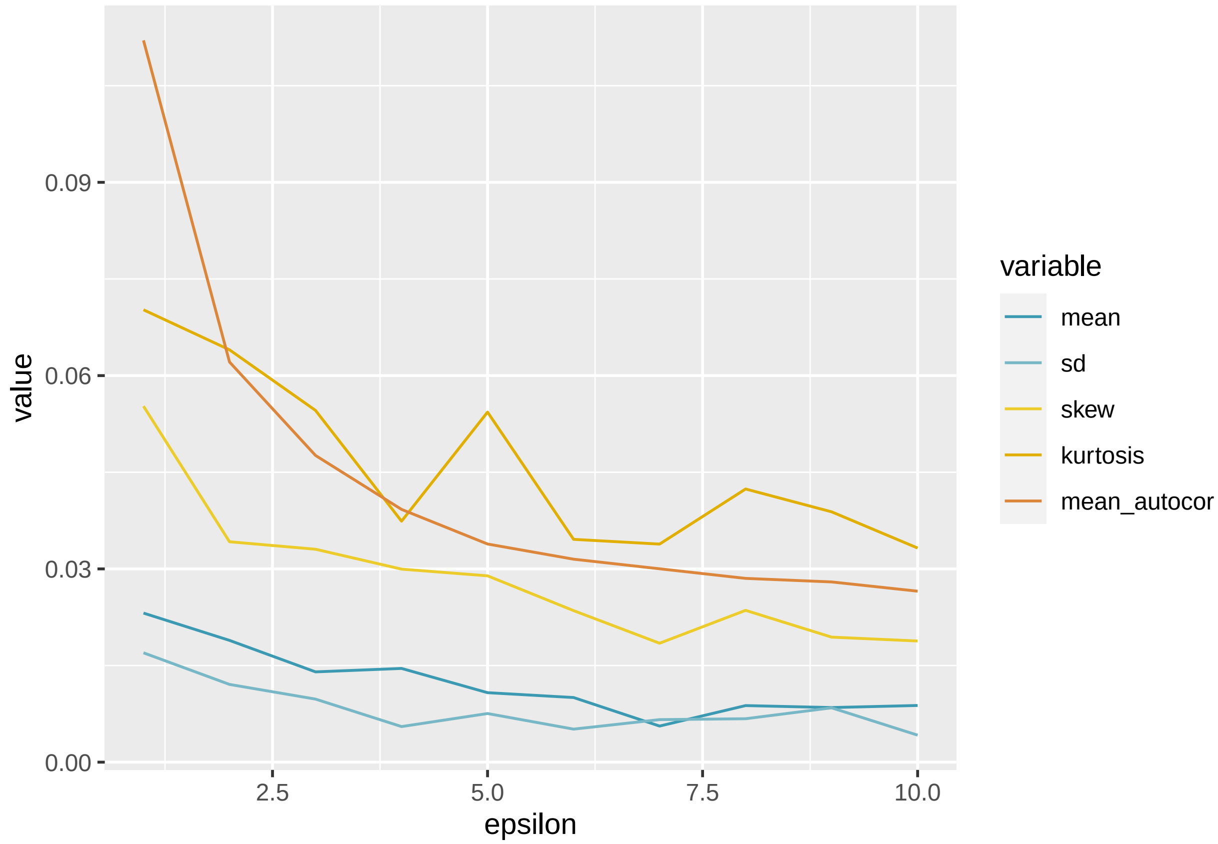
Nouvelle approche à venir.

3.3 Hasting iid

Afin de pouvoir appliquer numériquement la méthode de Reverse logistic regression, on aurait besoin de savoir simuler un échantillon iid suivant une loi dont on ne connait pas la constante de normalisation. L'idée est d'utiliser l'algorithme d'Hasting et de ne conserver qu'un échantillon tous les ϵ pas. Le code est en haut de ce document.

Etude de l'impact du choix du pas.

```
ggplot(df_hasting_iid_melted, aes(x = epsilon, y = value)) + geom_line(aes(color = variable, group = var
```




```
#ggplot(df_hasting_iid, aes(x = epsilon, y = time)) + geom_line()
```

$\epsilon = 2$ semble être un bon choix au regard du gain en terme d'auto-corrélation et du temps de calcul.

3.4 Reverse logistic regression : deux lois de même famille

```
pm_barre = function(u, theta){
  return(exp(-0.5 * ((u - theta[1]) / theta[2]) ** 2))
}

m = 1000
n = 10000
x = rnorm(m,2,4)
psi = c(mean(x), sd(x))

alpha = mc_mle(x, n, psi, pm_barre)
print(alpha)
```

```
## [1] 1.619072 4.009110
```

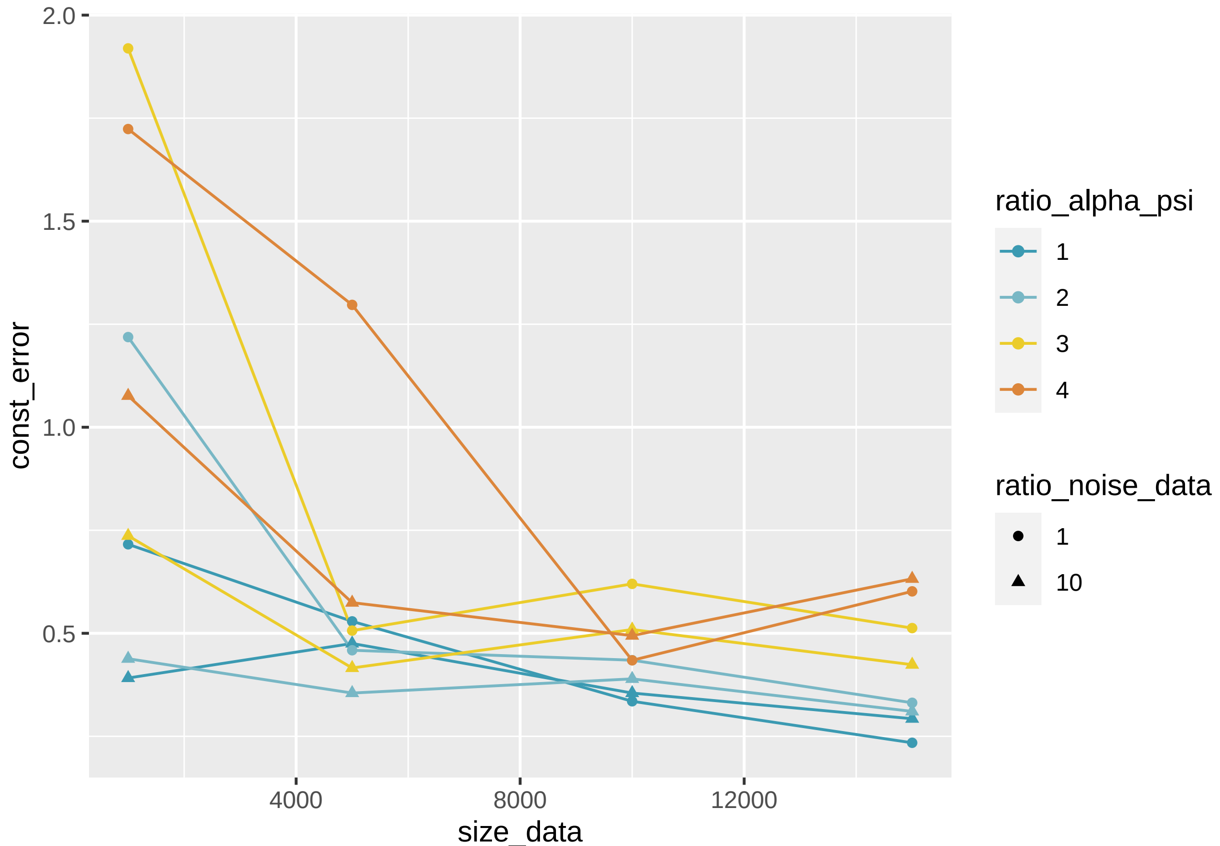
```
print(rev_log_reg(x, alpha, c(8,8), n, pm_barre, 2))
```

```
## [1] 9.18565
```

Etude de l'impact de la dimension, du ratio et des paramètres sur la convergence de la constante.

```
df_rev_log_reg <- read_csv("df_rev_log_reg.csv")[,-1]
```

```
df_rev_log_reg_agg = aggregate(const ~ size_data + ratio_noise_data + ratio_alpha_psi,
                    data = df_rev_log_reg,
                    FUN = mean)
df_rev_log_reg_agg$ratio_alpha_psi = as.factor(df_rev_log_reg_agg$ratio_alpha_psi)
df_rev_log_reg_agg$ratio_noise_data = as.factor(df_rev_log_reg_agg$ratio_noise_data)
df_rev_log_reg_agg$size_data = as.numeric(df_rev_log_reg_agg$size_data)
df_rev_log_reg_agg$const_error = abs(df_rev_log_reg_agg$const - 4*sqrt(2*pi))

ggplot(df_rev_log_reg_agg, aes(x = size_data, y = const_error, color = ratio_alpha_psi, shape = ratio_no
```



3.5 Reverse logistic regression : deux lois de familles différentes

```
pm_barre = function(u, theta){
  return(exp(-0.5 * ((u - theta[1]) / theta[2]) ** 2))
}

m = 1000
n = 10000
x = rnorm(m,2,4)
psi = c(mean(x), sd(x))

alpha = mc_mle(x, n, psi, pm_barre)
print(alpha)
```

```
## [1] 2.160142 3.978424
```

```r
rev_log_reg_with_noise(x, c(2,4), c(mean(x),sd(x)), n, pm_barre, dcauchy, rcauchy)
```

```
## [1] 10.03752
```

```r
df_rev_log_reg_noise <- read_csv("df_rev_log_reg_noise.csv")[,-1]

df_rev_log_reg_agg = aggregate(const ~ size_data + ratio_noise_data + law_noise,
                    data = df_rev_log_reg_noise,
                    FUN = mean)
df_rev_log_reg_agg$ratio_noise_data = as.factor(df_rev_log_reg_agg$ratio_noise_data)
df_rev_log_reg_agg$size_data = as.numeric(df_rev_log_reg_agg$size_data)
df_rev_log_reg_agg$const_error = abs(df_rev_log_reg_agg$const - 4*sqrt(2*pi))

ggplot(df_rev_log_reg_agg, aes(x = size_data, y = const_error, color = law_noise, shape = ratio_noise_da
```